# Elements of Programming Languages
# Tutorial 4: Subtyping and polymorphism
# Solution notes

1. **Subtyping and type bounds**

   (a)

   $$Sub1 <: Super \qquad Sub2 <: Super$$

   (b) i. $Sub1 \times Sub2 <: Super \times Super$ This holds:

   $$\frac{\overline{Sub1 <: Super} \quad \overline{Sub2 <: Super}}{Sub1 \times Sub2 <: Super \times Super}$$

   ii. $Sub1 \rightarrow Sub2 <: Super \rightarrow Super$ This does not hold since $Super <: Sub1$ doesn't.

   $$\frac{\overset{???}{Super <: Sub1} \quad \overline{Sub2 <: Super}}{Sub1 \rightarrow Sub2 <: Super \rightarrow Super}$$

   iii. $Super \rightarrow Super <: Sub1 \rightarrow Sub2$ This does not hold since $Super <: Sub2$ doesn't.

   $$\frac{\overline{Sub1 <: Super} \quad \overset{???}{Super <: Sub2}}{Super \rightarrow Super <: Sub1 \rightarrow Sub2}$$

   iv. $Super \rightarrow Sub1 <: Sub2 \rightarrow Super$ This holds:

   $$\frac{\overline{Sub1 <: Super} \quad \overline{Sub2 <: Super}}{Super \rightarrow Sub1 <: Sub2 \rightarrow Super}$$

   v. $(\star)$ $(Sub1 \rightarrow Sub1) \rightarrow Sub2 <: (Super \rightarrow Sub1) \rightarrow Super$ This holds:

   $$\frac{\dfrac{\overline{Sub1 <: Super} \quad \overline{Sub1 <: Sub1}}{Super \rightarrow Sub1 <: Sub1 \rightarrow Sub1} \quad \overline{Sub2 <: Super}}{(Sub1 \rightarrow Sub1) \rightarrow Sub2 <: (Super \rightarrow Sub1) \rightarrow Super}$$

   (c) If we call `f1` on `Sub2(true)` then the result has type `Super`. We can't access the `b` field because of a type mismatch.

   (d) This typechecks, because in either case we return `x` which has type `A`. If we apply it to a value of type `Sub1` or `Sub2` we get the same value back. If we apply it to `42 : Int` then we get a match error.

   (e) This typechecks, because as for `f2` we return `x : A` in either case. However, now if we apply to `Sub1` or `Sub2` we get the same value back, while if we apply to something of an unrelated type we get a type error. This seems to solve the problem.

2. **Subtyping and Contravariance**

   (a) `f` could call its function argument on any `Shape`, e.g. either `Circle` or `Rectangle`. Thus, calling `f` on a function of type `Rectangle => Int` is not allowed, because `Rectangle => Int` is not a subtype of `Shape => Int`. If this call was executed, then `f` could call its argument on a `Circle`, which would not match the expected `Rectangle` argument type.

(b) `g` can only call its function argument on a `Circle`. Thus, calling `g` on a function of type `Shape => Int` is allowed, because `Shape => Int` is a subtype of `Circle => Int`. If we execute this call, then whatever `g` does with its function argument will be fine, since the expected type of the function argument is `Shape`, so it can handle any particular type of shape such as `Circle`.

3. **Type parameters**

(a)

```
abstract class Tree[A]
case class Leaf[A](a: A) extends Tree[A]
case class Node[A](t1: Tree[A], t2: Tree[A]) extends Tree[A]
```

(b)

```
def sum(t: Tree[Int]) : Int = t match {
  case Leaf(a) => a
  case Node(t1,t2) => sum(t1) + sum(t2)
}
```

(c)

```
def map[A,B](t: Tree[A])(f: A => B): Tree[B] = t match {
  case Leaf(a) => Leaf(f(a))
  case Node(t1,t2) => Node(map(t1)(f), map(t2)(f))
}
```

(d)

```
def flatten[A](t: Tree[Tree[A]]): Tree[A] = t match {
  case Leaf(u) => u
  case Node(t1,t2) => Node(flatten(t1),flatten(t2))
}
```

(e)

```
def flatMap(t: Tree[A])(f: A => Tree[B]) = flatten(map(t)(f))
```